# 6. 记忆

## 6.1 记忆概述

想一想，我们为什么需要记忆 ？

大语言模型本质上是无记忆的。当我们与其交互时，它仅根据提供的提示生成相应的输出，而无法存储或记住过去的交互内容。这一特性，使得大语言模型在实现聊天机器人或聊天代理时，难以满足人们的期望。

人们期待聊天机器人具有人的品质和回应能力。当他们意识到机器人只是进行一次性的调用和响应，而无法记住以往的交流内容时，可能会感到沮丧。在现实的聊天环境中，人们的对话中充满了缩写和含蓄表达，他们会引用过去的对话内容，并期待对方能够理解和回应。例如，如果聊天历史中只在一开始提到了某人的名字，随后仅用代词指代，那么他们就期望聊天机器人能够理解和记住这个指代关系。

我们对聊天机器人的期待并不仅仅是其具备基础的回答功能，我们希望机器人能够在整个对话过程中，理解我们的话语，记住我们的交流内容，甚至理解我们的情绪和需求。为了实现这个目标，我们需要赋予大语言模型一种“记忆”能力。

记忆是一种基础的人类特性，是我们理解和交流世界的基础。因此，我们将这种能力赋予机器人，使其具备人类般的记忆功能。这种记忆功能被我们封装在记忆模块和记忆组件中，它们不仅能够存储和管理信息，还能够根据需要提取和使用这些信息。当我们谈论内存时，我们仅仅是在谈论一个电脑的内存条。但如果我们还谈论一种理解和交流的能力，一种建立和维持关系的能力，我们更愿意使用“记忆”。我们希望机器人能够像人一样理解我们的话语，记住我们的交流内容，甚至理解我们的情绪和需求。

所以，我们将 Memory 模块称之为记忆模块，各种封装的类和示例，我们称为记忆组件。因为我们不会说自己有内存，我们说自己有记忆。

最后，我们要强调为什么在实际的大语言模型应用开发中，需要记忆模块的功能。简单来说，同样作为提示词模板的外部数据，“记忆”的功能形成的外部数据，与检索的外部文档内容起到一样的作用。同样地，可以确保大语言模型在处理信息时始终能够获取到最新、最准确的上下文数据；通过提供聊天信息，我们可以让大模型的输出更有据可循，多了一份“证据”；这也是一种低成本应用大语言模型的策略，不需要做涉及参数训练等额外工作。

大语言模型应用落地遇到的问题

记忆组件的工作原理相当直观：它接收聊天消息，将这些消息作为提示词模板的一部分，然后传递给模型I/O模块的模型包装器。随后，模型将根据这些提示词的约束，生成相应的输出。我们在模型I/O模块的提示词模板中有提到，将提示词模板的数据分为内部和外部数据，聊天信息是发生在应用程序本身的，属于内部数据。在数据连接模块，通过 LEDVR 工作流提取的相关性文档信息，属于外部数据。

你可能会注意到，当聊天信息和文档信息，所有这些信息都会被存储在“提示词模板”中。然而，这种方法在某些情况下会出现问题，那就是当我们进行长期的对话时，大语言模型可能无法容纳所有的信息，每个大语言模型都有 Max Tokens 的限制。即使是强大的 OpenAI 平台，它的最新模型 GPT-4-32k-0613 在一次处理中,Max Tokens 是 32768。但是聊天记录这在许多应用中都是一个巨大的信息量，它可以包含长篇的文章、复杂的对话内容，或者大量的其他类型的数据。比如一个翻译语言的应用，这些聊天记录会变得很大很大。然而，尽管如此，由于 Max Tokens 的限制仍然无法保存完整的对话内容，尤其是在持续的、涉及大量交互的对话场景中。

这个问题引发了一系列的挑战：我们如何有效地管理这个Max Tokens，适应各个模型平台的 API 要求？如何决定在每个会话中使用哪些聊天记录，以及如何存储和检索过去的对话内容？这就是我们需要记忆组件的地方。比如对话总结记忆组件就提供了一个解决方案。这种记忆模式会在对话进行的过程中实时总结对话内容，并将当前的摘要存储在应用程序运行时内存中。这种记忆方式对于长期的对话来说尤其有用，因为如果直接在提示中保留过去的消息历史。

## 6.1.1 记忆组件的定义

记忆组件是什么？

记忆组件，实际上是一个聊天备忘录，像苹果手机备忘录程序一样，我们可以用它记录我们与大语言模型的聊天对话消息。那么，备忘录的作用是什么呢？试想在一个会议场合，你有一位秘书在边上为你做备忘录。当你发言时，你可能会忘记先前的发言者都说过什么，这时你可以让秘书展示备忘录，通过查阅这些信息，你能够清楚地整理自己的发言，从而赢得全场的赞许。而当你发言结束后，秘书又要做什么呢？他需要把你刚刚的精彩发言记录下来。

这就是记忆组件的两大功能：读取和写入。因此，记忆组件的基本操作包括读取和写入。在链组件的每次运行中，都会有两次与记忆组件的交互：在执行核心逻辑之前读取记忆组件以丰富用户输入的上下文语境，在执行核心逻辑后将当前运行的输入和输出写入记忆组件，以便在未来的运行中引用。

记忆组件的设计，旨在解决两个核心问题：一是如何写入，也就是存储状态，二是如何读取，即查询状态。状态存储一般通过程序运行的内存存储历史聊天记录实现。状态查询则依赖于在聊天记录上构建的数据结构和算法，它们为我们提供了对历史消息的高效检索和解析。类似于一个聊天备忘录，记忆组件既可以帮我们记录下聊天的每一条信息，也可以依据我们的需求，搜索出相关的历史聊天记录。

最难理解的是如何在聊天记录上构建的数据结构和算法？简单来说，数据结构和算法就是用来整理和检索聊天记录的方法。记忆组件会将处理过的聊天信息数据注入到提示词模板中，最终通过模型平台的API接口获取大语言模型的响应，从而使得这个响应更加准确。我们需要决定哪些聊天记录需要保留，哪些聊天记录需要进一步处理，这个过程就是在聊天记录上构建数据结构和算法。

我们主要采取以下两种方法整理和检索聊天记录。一种是全部放入提示词模板中，这是一种比较简单的方法，将聊天窗口上下文直接放在提示词中，作为外部数据注入到提示词模板，再提供给大语言模型。这种方法简洁易行，但是因为结合方式较为粗糙，所以可能无法达到精准控制的目的。而且由于模型平台的 Max Tokens 限制，这种方式要考虑的是如何截取聊天记录，如何压缩聊天记录，适应模型平台底层的API要求。

第二种方法是压缩后再放入提示词模板中。类似计算机的文件和图像压缩技术。这种方法的思路是多一次，甚至多几次先把聊天记录信息提交给大语言模型做总结，或者从数据库中提取相关的聊天记录，提取实体知识等操作，然后将这些文本拼接在提示词中，再提供给大语言模型，进行下一轮对话。

这个过程中，我们可以使用类似于图像压缩算法的方式对聊天记录进行压缩。比如，我们可以参考 JPEG 图像压缩算法的原理，将原始的聊天记录（类比为图像的原始数据）进行压缩。JPEG 图像压缩算法在压缩过程中会舍弃一些人眼难以察觉的高频细节信息，从而达到压缩的效果，同时大部分重要信息会被保留。这样，我们就可以将压缩后的聊天记录（类比为压缩后的图像）放入提示词模板中，用于生成下一轮对话。虽然这种压缩方式可能会丢失一些细节信息，但是由于这些信息对于聊天记录的总体含义影响较小，因此可以接受。

通过这种方式，我们不仅可以在保证聊天记录的总体含义的同时，减小提示词的长度，而且还可以提高大语言模型的运行效率和响应速度。

## 6.1.2 记忆、链和Agent的关系

我们先探讨记忆组件和链组件的关系。LangChain 的记忆模块提供了各种类型的记忆组件，这些记忆组件可以独立使用，也可以无缝地集成到链组件中。换句话说，记忆组件可以作为链组件的一个部分，是一个内部组件，用于保存状态和历史信息，以便链组件在处理输入和生成输出时使用。所以记忆组件和链组件的关系是，链组件是与大语言模型交互的主体，而记忆组件则是在这个过程中提供支持的角色。通过引入记忆组件，链组件调用大语言模型得到能“回顾”过去的对话，理解并确定当前谈论的主题和对象。这使得聊天机器人更接近人类的交流方式，从而更好地满足用户的期望。记忆组件最常见的使用场景是智能助理的聊天界面，在这里，它扮演着不可或缺的角色，提升了大语言模型的对话能力和用户体验。

我们更进一步探讨记忆组件和Agent的关系。在Agent模块中，我们将记忆组件放入智能体Agent中。这使得Agent不仅能够处理和响应我们的请求，而且还能够记住我们的交流内容。这就是我们所说的“让Agent拥有记忆”。 Agent是一个更高级别的组件，它可能包含一个或多个链组件，以及与这些链组件交互的记忆组件。Agent通常代表一个完整的对话系统或应用，负责管理和协调其内部组件（包括链和记忆组件）以响应用户输入。

具体来说，智能体Agent在处理用户输入时，可能会使用其内部的链组件来执行各种逻辑，同时使用内存来存储和检索过去的交互信息。例如，Agent可能会使用内存中的信息来增强用户输入，或者将链组件的输出写入内存以供未来的交互中引用。因此，我们可以说记忆组件是智能体 Agent 的一个重要部分，帮助 Agent 维护状态和历史信息，使其能够处理复杂的对话和任务。在构建智能体 Agent 时，通常需要考虑如何选择和配置内存，以满足特定的应用需求和性能目标。

通过赋予Agent记忆，我们让机器人变得更加人性化，更加接近我们的交流方式。这不仅能够提高机器人的交流效率和质量，还能够增强我们对机器人的信任和满意度。因此，记忆不仅仅是一种技术，更是一种理念，一种让机器人更接近人类的方式。将记忆组件放入智能体Agent中，“让Agent拥有记忆”，也是我们在本章中关注的地方。

这就是记忆组件在 LangChain 中的运作方式：集成到链组件中，集成到智能体Agent中。

## 6.1.3 第一个记忆组件

我们可以用代码演示如何实例化一个记忆组件，以及如何使用记忆组件的保存和加载方法来管理聊天记录。这种使用记忆组件的方式对于所有的记忆组件都是通用的，只是在具体的实现细节上可能会有所不同。

首先，我们需要引入 ConversationTokenBufferMemory 类，并创建一个 OpenAI 类的实例作为其参数。 ConversationTokenBufferMemory 是一个记忆组件，它将最近的交互信息保存在内存中，并使用令牌长度而不是交互次数来决定何时清除交互信息。

from langchain.memory import ConversationTokenBufferMemory

from langchain.llms import OpenAI

openai\_api\_key = "填入你的OpenAI密钥"

llm = OpenAI(openai\_api\_key=openai\_api\_key)

memory =

ConversationTokenBufferMemory(llm=llm,max\_token\_limit=1000,memory\_key="session\_chat")

在上面的代码中，我们首先创建了一个 OpenAI 实例 llm 类型的模型包装器。

然后用这个实例和一个max\_token\_limit参数来创建 ConversationTokenBufferMemory 的实例 memory 。max\_token\_limit 参数用于设置记忆组件可以保存的 Max Token 数量, 数字设置过于小的话，可能最后打印的对象是空字符串，可以手动设置 10，再设置 100 查看效果。 memory\_key 参数用于设置记忆组件存储对象的键名，默认是 history ，这里我们设置为 session\_chat 。

接下来，我们可以使用 save\_context 方法来将聊天记录保存到记忆组件中。每次调用 save\_context ，都会将一次交互（包括用户的输入和大语言模型的输出）添加到记忆组件的缓冲区中。

memory.save\_context({"input": "你好！我是李特丽，这是人类的第一个消息"}, {"output": "你好！我是AI助理的第一个消息"})

memory.save\_context({"input": "今天心情怎么样"}, {"output": "我很开心认识你"})

在上面的代码中，我们先后保存了两次交互记录。第一次交互中，用户的输入是 "你好！我是李特丽，这是人类的第一个消息"；第二次交互中，用户的输入是 "今天心情怎么样"，大语言模型的输出是 "我很开心认识你"。

memory.load\_memory\_variables({})

最后，我们可以使用 load\_memory\_variables 方法来加载记忆组件中的聊天记录。这个方法会返回一个字典，其中包含了记忆组件当前保存的所有聊天记录。

{'session\_chat': 'Human: 你好！我是李特丽，这是人类的第一个消息\nAI: 你好！我是AI助理的第一个消息\nHuman: 今天心情怎么样\nAI: 我很开心认识你'}

以上就是如何使用 ConversationTokenBufferMemory 记忆组件的示例。

## 6.1.4 内置记忆组件

Langchain 的记忆模块( langchain.memory 包)提供了多种内置的记忆组件类，这些类的使用方法和 ConversationTokenBufferMemory 基本一致，都提供了保存（ save\_context ）和加载（ load\_memory\_variables ）聊天记录的方法，只是在具体的实现和应用场景上有所不同。

例如， ConversationBufferMemory 、 ConversationBufferWindowMemory 、 ConversationTokenBufferMemory 等类都是用于保存和加载聊天记录的记忆组件，但它们在保存聊天记录时所使用的数据结构和算法有所不同。 ConversationBufferMemory 使用一个缓冲区来保存最近的聊天记录，而 ConversationTokenBufferMemory 则使用 Max Tokens 长度来决定何时清除聊天记录。

此外，有些记忆组件类还提供了额外的功能，如 ConversationEntityMemory 和 ConversationKGMemory 类可以用于保存和查询实体信息， CombinedMemory 类可以用于组合多个记忆组件，而 ReadOnlySharedMemory 类则提供了一种只读的共享记忆模式。

对于需要将聊天记录持久化保存的应用场景，Langchain 的记忆模块还提供了多种与数据库集成的记忆组件类，如 SQLChatMessageHistory 、 MongoDBChatMessageHistory 、 DynamoDBChatMessageHistory 等。这些类在保存和加载聊天记录时，会将聊天记录保存在对应的数据库中。

在选择记忆组件时，你需要根据自己的应用需求来选择合适的类。例如，如果你的应用需要处理大量的聊天记录，或者需要在多个会话中共享聊天记录，那么你可能需要选择一个与数据库集成的记忆组件。而如果你的应用主要处理实体信息，那么你可能需要选择 ConversationEntityMemory 或 ConversationKGMemory 这样的类。无论你选择哪种记忆组件，都需要理解其工作原理和使用方法，以便正确地使用它来管理你的聊天记录。

## 6.1.5 自定义记忆组件

我们在内置记忆组件一节中，了解了 Langchain 的内置记忆组件，尽管这些组件能够满足许多通用的需求，但每个具体的应用场景都有其独特的要求和复杂性。例如，某些应用可能需要特定的数据结构来优化查询性能，或者需要特别的存储方式来满足数据安全或隐私要求。在这些情况下，预设的记忆组件可能无法完全满足需求。通过允许开发者添加自定义的记忆类型，框架能够提供更高的灵活性和扩展性，使得开发者能够根据自己的需求定制和优化记忆组件。这对于高级的使用场景，如大规模的生产环境或特定的业务需求，尤为重要。

对于初学者来说，创建自定义组件实际上是理解和掌握框架的有效路径。这是因为在构建自定义组件的过程中，你将被迫深入理解框架的设计逻辑、工作方式以及内置组件的实现机制。这将极大地加深你对框架整体架构和工作原理的理解。

内置组件往往是框架开发者根据通用需求预先设计和封装好的，使用者可以直接拿来使用，无需关心其内部实现细节。这无疑大大降低了使用门槛，提高了开发效率。然而，这也意味着使用者很可能对这些组件的内部构造和工作原理一无所知。

而当你试图创建自定义组件时，你需要深入理解和分析你的特定需求，然后再在此基础上设计和实现符合需求的组件。这个过程将迫使你深入了解内置组件的构造和工作原理，从而更好地理解框架的设计逻辑和工作方式。因此，尽管创建自定义组件的过程可能会有些复杂和困难，但它无疑是深入理解和掌握框架的有效方式。

对于 LangChain 框架的设计者来说，或者站在在 LangChain 立场来说，这样的设计都会提供自定义记忆组件的示例代码和测试代码示例，目的是确保开发者能够更容易地理解和使用框架，从而降低开发门槛。这对于提高框架的用户友好性和推动其广泛应用至关重要。另外一个考虑是框架的设计者深知不可能预设所有可能的需求。在实际的应用场景中，每个项目或业务都有其特定的需求，各行各业需求不一样，所以 LangChain 先满足的是通用的场景，比如记忆模块的内置组件，多数都是会话记忆组件，而且会将会话记忆组件分很多种类，满足这个场景下的各种需求 。但是并不会满足任何的场景，比如金融、医疗这些细分行业。这样的设计理念，也是 LangChain 框架在满足通用需求的同时，不断推动创新和适应复杂应用场景的关键策略。

在本节中，我们将向 ConversationChain 添加一个自定义的记忆类型。为了添加自定义的记忆类，我们需要引入基础记忆类并对其进行子类化。请注意，这种实现方式相当简单且脆弱，可能在生产环境中并不实用。它的目的是展示你可以添加自定义记忆实现。

from langchain import OpenAI, ConversationChain

from langchain.schema import BaseMemory

from pydantic import BaseModel

from typing import List, Dict, Any

在这个例子中，我们将编写一个自定义记忆类，它使用 spacy 来提取实体，并将有关它们的信息保存在一个简单的哈希表中。然后，在对话中，我们将观察输入文本，提取实体，并将关于它们的任何信息放入上下文中。

!pip install spacy

!python -m spacy download en\_core\_web\_lg

import spacy

nlp = spacy.load("en\_core\_web\_lg")

class SpacyEntityMemory(BaseMemory, BaseModel):

"""Memory class for storing information about entities."""

# Define dictionary to store information about entities.

entities: dict = {}

# Define key to pass information about entities into prompt.

memory\_key: str = "entities"

def clear(self):

self.entities = {}

@property

def memory\_variables(self) -> List[str]:

"""Define the variables we are providing to the prompt."""

return [self.memory\_key]

def load\_memory\_variables(self, inputs: Dict[str, Any]) -> Dict[str, str]:

"""Load the memory variables, in this case the entity key."""

# Get the input text and run through spacy

doc = nlp(inputs[list(inputs.keys())[0]])

# Extract known information about entities, if they exist.

entities = [

self.entities[str(ent)] for ent in doc.ents if str(ent) in self.entities

]

# Return combined information about entities to put into context.

return {self.memory\_key: "\n".join(entities)}

def save\_context(self, inputs: Dict[str, Any], outputs: Dict[str, str]) -> None:

"""Save context from this conversation to buffer."""

# Get the input text and run through spacy

text = inputs[list(inputs.keys())[0]]

doc = nlp(text)

# For each entity that was mentioned, save this information to the dictionary.

for ent in doc.ents:

ent\_str = str(ent)

if ent\_str in self.entities:

self.entities[ent\_str] += f"\n{text}"

else:

self.entities[ent\_str] = text

我们现在定义一个提示词模板，它接受关于实体的信息以及用户输入的信息。

from langchain.prompts.prompt import PromptTemplate

template = """The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

{entities}

Conversation:

Human: {input}

AI:"""

prompt=PromptTemplate(input\_variables=["entities","input"], template=template)

然后我们把记忆组件和链组件组合在一起，在 ConversationChain 链上运行，与大语言模型交互。

llm = OpenAI(temperature=0)

conversation = ConversationChain(

llm=llm, prompt=prompt, verbose=True, memory=SpacyEntityMemory()

)

在第一个例子中，由于没有关于 Harrison 的先验知识，"Relevant entity information"（相关实体信息）部分是空的。

conversation.predict(input="Harrison likes machine learning")

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

Conversation:

Human: Harrison likes machine learning

AI:

> Finished ConversationChain chain.

" That's great to hear! Machine learning is a fascinating field of study. It involves using algorithms to analyze data and make predictions. Have you ever studied machine learning, Harrison?"

现在在第二个例子中，我们可以看到它抽取了关于 Harrison 的信息。

conversation.predict(

input="What do you think Harrison's favorite subject in college was?"

)

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. You are provided with information about entities the Human mentions, if relevant.

Relevant entity information:

Harrison likes machine learning

Conversation:

Human: What do you think Harrison's favorite subject in college was?

AI:

> Finished ConversationChain chain.

' From what I know about Harrison, I believe his favorite subject in college was machine learning. He has expressed a strong interest in the subject and has mentioned it often.'

请再次注意，这种实现方式相当简单且脆弱，可能在生产环境中并不实用。它的目的是为了展示你可以添加自定义的记忆组件。

## 6.2 记忆增强检索能力的实践

本节我们将深入探讨如何通过集成记忆组件，增强我们的数据连接模块LEDVR工作流中的数据检索能力，从而提升QA问答应用的回答质量。我们将以《链》章节中的代码案例为基础，介绍如何在已有的检索器基础上，添加记忆组件，使ConversationalRetrievalChain链组件具有“记忆”能力。

在此基础上，我们的案例应用将得以进一步提升。以LUA语言开发者在线文档问答为例，通过增强后的链组件，我们的程序不仅能在 http://developers.mini1.cn/wiki/luawh.html 中找到相关的文档内容，准确回答如“LUA宿主语言是什么”等问题，更能够记住用户在对话中的相关信息，如用户的名字，从而对更多复杂的问题提供更准确的答复。例如，当用户在问完几轮关于LUA语言编程的问题之后，如果他们再次提到他们的名字或引用他们之前提到的信息，增强后的ConversationalRetrievalChain链组件能够记住并理解这些信息，提供更加精确和个性化的回答。

希望读者能够理解并掌握如何将记忆组件与链组件集成，从而实现数据检索能力的增强，提升QA问答应用的回答质量。同时，我们也将向读者展示如何通过增加记忆组件，使我们的程序更具人性化，能够更好地满足用户的需求。使用加载器开始，到创建向量存储库实例都与《链》章节一样。如果你已经很熟悉这段代码，可以直接跳过这部分。

## 6.2.1 LEDVR 获取外部数据

首先，我们需要从网络加载文档。这可以通过使用WebBaseLoader来完成：

from langchain.document\_loaders import WebBaseLoader

openai\_api\_key="填入你的密钥"

loader = WebBaseLoader("http://developers.mini1.cn/wiki/luawh.html")

data = loader.load()

你也可以选择其他的加载器和其他的文档资源。接下来，我们需要创建一个嵌入模型实例的包装器，这可以使用OpenAIEmbeddings完成：

from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings(openai\_api\_key=openai\_api\_key)

然后，我们需要将文档切割成块，这可以通过使用RecursiveCharacterTextSplitter来完成：

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=0)

splits = text\_splitter.split\_documents(data)

接着，我们需要创建一个向量存储库，这里我们选择使用FAISS，现在我们有了一个向量存储库：

from langchain.vectorstores import FAISS

vectordb = FAISS.from\_documents(documents=splits,embedding=embedding)

## 6.2.2 加入记忆组件

我们首先导入 ConversationBufferMemory 类，这是最常见的记忆组件，其工作原理非常简单：仅将所有聊天记录保存起来，而不使用任何算法进行截取或提炼压缩。在提示词模板中，我们可以看到所有的聊天记录。

from langchain.memory import ConversationBufferMemory

memory= = ConversationBufferMemory(memory\_key="chat\_history",return\_messages=True)

初始化记忆组件后，我们可以看到其内部的存储情况。由于我们刚刚进行了初始化，所以存储的聊天记录为空。然后，我们通过 add\_user\_message 方法添加一个人类的消息，向程序介绍我的名字。此时，我们再次查看记忆组件，就可以看到添加了一条 HumanMessage 消息。我们看看初始化后，记忆组件里保存着什么记录。因为我们刚刚初始化，所以是[]。

# 打印 memory.chat\_memory.messages

[]

我们使用 add\_user\_message 添加一个人类的消息，向程序介绍我的名字。

memory.chat\_memory.add\_user\_message("我是李特丽")

第一次打印 memory.chat\_memory.messages 时为[], 当我推送一条自我介绍后，我们可以看到添加了一条 HumanMessage 消息。

[HumanMessage(content='我是李特丽', additional\_kwargs={}, example=False)]

使用 memory 组件的 load\_memory\_variables 方法可以看保存在程序运行内存中的 memory 对象，主键是 chat\_history，正如我们初始化 ConversationBufferMemory 设置的那样：memory\_key="chat\_history" 。

# 打印memory.load\_memory\_variables({})

{'chat\_history': [HumanMessage(content='我是李特丽', additional\_kwargs={}, example=False)]}

我们采用最直接的方式来演示记忆组件是如何与链组件共同工作的。从提示词模板出发，逐步增加组件的方式，展示这个工作机制。这一次我们使用的链组件是 load\_qa\_chain。这个链组件的优势在于专门用于QA问答，对于合并文档链组件不必掌握这些链的使用方法，只要在这个链上指定类型。导入 load\_qa\_chain 、ChatOpenAI 和 PromptTemplate。提示词模板用于我们初始化 load\_qa\_chain 的时候，个性化配置提示词。

from langchain.chat\_models import ChatOpenAI

from langchain.chains.question\_answering import load\_qa\_chain

from langchain.prompts import PromptTemplate

# 这里需注意使用 Chat Model 类型的模型包装器，并且使用先进的模型型号。

llm = ChatOpenAI(openai\_api\_key=openai\_api\_key,temperature=0, model="gpt-3.5-turbo-0613")

使用向量存储库实例 的 similarity\_search 方法，测试是否可以检索到与问题相关的文档了。我们可以打印 len(docs)， 看看这个问题搜索到了几个文档片段。检索到的相关文档内容，我们要输入到提示模板包装器中，这一步先测试向量存储库是否正常工作。

query = "LUA 的宿主语言是什么？"

docs = vectordb.similarity\_search(query)

docs

打印 len(docs) 的长度是4，有4个相关文档片段被检索到。

4

创建提示词是最重要的环节，在创建的过程中我们可以理解为什么加入记忆后，因为有了“聊天备忘录” 记录的内容，让链组件有了 “记忆”。使用提示模板包装器，我们自定义一个提示词模板字符串。

提示词内容分为四部分：一是对模型的指导词：“请你回答问题的时候，请依据文档内容和聊天记录回答，如果在其中找不到相关信息或者答案，请回答不知道。” ； 二是用问题检索到的相关文档内容：“文档内容是：{context}” ；三是记忆组件输出的记忆内容：“聊天记录是：{chat\_history}” ；四是用户的输入： “Human: {human\_input}”。

template = """你是说中文的chatbot.

请你回答问题的时候，请依据文档内容和聊天记录回答，如果在其中找不到相关信息或者答案，请回答不知道。

文档内容是：{context}

聊天记录是：{chat\_history}

Human: {human\_input}

Chatbot:"""

prompt = PromptTemplate(

input\_variables=["chat\_history", "human\_input", "context"], template=template

)

记忆组件除了指定记忆存储对象的键值，还要指定 input\_key 。load\_qa\_chain 链组件在运行时，会解析 input\_key ，将值对应到模板字符串的用户输入 human\_input 占位符中。

memory=ConversationBufferMemory(

memory\_key="chat\_history", input\_key="human\_input")

chain = load\_qa\_chain(

llm=llm, chain\_type="stuff", memory=memory, prompt=prompt

)

刚刚我们把记忆组件加入到 load\_qa\_chain 链组件中，这个链组件就有了记忆。我们向这个链组件发出第一个问题: “ LUA 的宿主语言是什么？”

query = "LUA 的宿主语言是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

不出意外，运行链组件后，我们得到了正确的答案。这个答案正是来源于我们之前检索的四个文档片段。

{'output\_text': 'Lua的宿主语言通常是C或C++。'}

接着我们可以相互之间来个自我介绍。

query = "我的名字是李特丽。你叫什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

大语言模型回答的是：“我是一个中文的chatbot。”

{'output\_text': '我是一个中文的chatbot。'}

我们继续模拟一下正常的聊天应用程序，问一些别的问题，目的是测试一下，几个问题后，他是否还能记得我们的名字，如果他能记得，则证明他有了记忆。我们先问技术文档上的问题。

query = "LUA 的循环语句是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

回答依然是准确的。

{'output\_text': 'Lua的循环语句有while循环、for循环和repeat...until循环。'}

现在我们可以测试他是不是记得我们的名字了。

query = "我的名字是什么？"

docs = vectordb.similarity\_search(query)

chain({"input\_documents":docs, "human\_input": query}, return\_only\_outputs=True)

他记得我们的名字了。

{'output\_text': '你的名字是李特丽。'}

我们打印看看他记忆了什么内容：

print(chain.memory.buffer)

显然，他把我们之间的聊天记录都记录下来了。用户的问题和模型的回答。

Human: LUA 的宿主语言是什么？

AI: Lua的宿主语言通常是C或C++。

Human: 我的名字是李特丽。你叫什么？

AI: 我是一个中文的chatbot。

Human: LUA 的循环语句是什么？

AI: Lua的循环语句有while循环、for循环和repeat...until循环。

Human: 我的名字是什么？

AI: 你的名字是李特丽。

在这个代码实践中，我们将记忆组件与 load\_qa\_chain 链组件一起使用，让这个链组件具有记忆能力。我们通过问一系列问题，观察链组件的回答，从而验证其记忆能力。例如，当我们问了几个关于 LUA 语言的问题之后，再问它我们的名字，它还能正确回答，这就证明了它具有记忆能力。

需要注意的是，这些聊天记录并不会一直保存，它们只保留在运行程序的内存中，一旦程序停止运行，这些记录就会消失。这个示例代码的目的是解释如何让一个 QA 问答链具备“记忆”的能力，实现检索增强。如果没有记忆的能力，对于用户来说，这个程序就会看起来很木讷，因为凡是“关于人”的问题，它回答的都是“不知道”。

总的来说，通过以上的代码实践，我们了解了如何使用 ConversationBufferMemory 这个最基本的记忆组件，包括如何实例化记忆组件，如何使用其存储和读取聊天记录的功能，以及如何将其与其他组件（例如 load\_qa\_chain 链组件）组合使用，来增强程序的功能。

## 6.3 记忆增强Agent能力的实践

Agent通常被赋予执行特定任务的能力，比如回答问题、进行对话，或者进行搜索等。然而，许多Agent在执行任务时可能遇到一个问题：它们无法“记住”先前的交互或信息。这就是我们希望通过“记忆增强”解决的问题。

记忆增强的目的是让Agent能够记住先前的交互或信息，使得它们能够在处理连续的任务或长期的交互时，利用历史信息进行更有效的决策。例如，一个能记住以前问题和答案的问答Agent，可以用这些记忆来更好地回答相关的后续问题。

在本文中，我们将探讨如何向Agent添加记忆组件，以实现记忆增强的效果。我们将通过具体的代码示例，介绍如何创建一个带有记忆的LLM链，并使用该链来创建一个自定义的Agent。我们还将介绍如何利用特定的工具（如Google搜索）来增强Agent的功能。

在这个过程中，我们希望能展示出记忆增强对于提升Agent性能的重要性，以及如何在实践中实现记忆增强。

## 6.3.1 向Agent添加记忆

我们将介绍如何向Agent 添加记忆组件。我们将执行以下步骤：创建一个带有记忆的LLM链；使用该LLM链创建一个自定义 Agent；我们将创建一个简单的自定义Agent，它具有访问搜索工具的权限，并使用ConversationBufferMemory类。

我们向 Agent添加了记忆组件。具体步骤如下：

首先，我们导入了所需的模块和类，包括 ZeroShotAgent 、 Tool 、 AgentExecutor 、 ConversationBufferMemory 、 ChatOpenAI 、 LLMChain 。

from langchain.agents import ZeroShotAgent, Tool, AgentExecutor

from langchain.memory import ConversationBufferMemory

使用 OpenAI，需要设置 openai\_api\_key 密钥，最好是使用 Chat Model 类的模型包装器 ChatOpenAI。

from langchain.chat\_models import ChatOpenAI

openai\_api\_key="填入你的OPENAI\_API\_KEY密钥"

llm = ChatOpenAI(openai\_api\_key=openai\_api\_key,temperature=0, model="gpt-3.5-turbo-0613")

from langchain.chains import LLMChain

使用 Google 搜索作为工具，需要设置 SERPAPI\_API\_KEY 密钥。

import os

os.environ["SERPAPI\_API\_KEY"] = "填入你的SERPAPI\_API\_KEY密钥"

设置代理类型为 ZERO\_SHOT\_REACT\_DESCRIPTION，加载所有的 tools，初始化这个代理。运行代理，我们询问“请告诉我OPENAI的CEO是谁？”。

from langchain.agents import initialize\_agent, load\_tools

from langchain.agents import AgentType

tools = load\_tools(["serpapi", "llm-math"], llm=llm)

agent = initialize\_agent(

tools,

llm,

agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,

)

agent.run("请告诉我OPENAI的CEO是谁？")

现在开始加入记忆组件。 Tools 对象将作为 Agent的一个工具，用于回答有关当前事件的问题可以借助的工具。我们指定了 LLMChain 作为链，并设置了 tools 作为工具。

接着，我们创建了一个 LLMChain 实例，并使用它来初始化 ZeroShotAgent 。在 LLMChain 的初始化中，我们指定了 ChatOpenAI 作为大语言模型，并设置了 prompt 作为提示词模板。在 ZeroShotAgent 的初始化中，我们指定了 LLMChain 作为链，并设置了 tools 作为工具。然后，我们将 ZeroShotAgent 和 tools 一起传入 AgentExecutor.from\_agent\_and\_tools 方法，创建了一个 AgentExecutor 实例。在这个过程中，我们还指定了 memory 作为记忆组件。最后，我们使用 AgentExecutor 实例的 run 方法，运行 Agent，并向其提出问题： "上海的人口是多少？"。

prefix = """你是一个说中文的chatbot,你可以使用tools帮你获得答案:"""

suffix = """你的中文回答是："

聊天记录：{chat\_history}

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt(

tools,

prefix=prefix,

suffix=suffix,

input\_variables=["input", "chat\_history", "agent\_scratchpad"],

)

memory = ConversationBufferMemory(memory\_key="chat\_history")

llm\_chain = LLMChain(llm=llm, prompt=prompt)

agent = ZeroShotAgent(llm\_chain=llm\_chain, tools=tools, verbose=True)

agent\_chain = AgentExecutor.from\_agent\_and\_tools(

agent=agent, tools=tools, verbose=True, memory=memory

)

agent\_chain.run("上海的人口是多少？")

> Entering new AgentExecutor chain...

Thought: 我需要搜索一下上海的人口数据。

Action: Search

Action Input: "上海人口数据"

Observation: 26.32 million (2019)

Thought:我现在知道了上海的人口是2632万（2019年）。

Final Answer: 上海的人口是2632万（2019年）。

> Finished chain.

'上海的人口是2632万（2019年）。'

为了测试加入记忆组件的 Agent 是否更加智能，第二个问题，使用代词“它”来迷惑大语言模型，就像我们跟朋友聊天时，我们谈论一个人，可能只会提及一次姓名。后面的聊天都会使用人称代词或者缩写，而不用每次都使用全名。

agent\_chain.run("它的地标建筑是什么？")

Entering new AgentExecutor chain...

Thought: 我需要搜索上海的地标建筑。

Action: Search

Action Input: 上海地标建筑

Observation: https://cn.tripadvisor.com/Attractions-g308272-Activities-c47-Shanghai.html

Thought:我现在知道上海的地标建筑了。

Final Answer: 上海的地标建筑包括东方明珠广播电视塔、外滩、上海博物馆等。

> Finished chain.

'上海的地标建筑包括东方明珠广播电视塔、外滩、上海博物馆等。'

为了对比，记忆组件在Agent组件中，发挥的什么作用？我们再创建一个Agent组件，但是不要加入记忆组件。我们使用相同的问题，测试Agent是否知道第二个问题“它的地标建筑是什么”，“它”指代的是“上海”。我们可以看到，虽然代码的大部分步骤都和之前一样，但是在创建 AgentExecutor 的时候，我们并没有指定 memory 参数。

prefix = """你是一个说中文的chatbot,你可以使用tools帮你获得答案:"""

suffix = """Begin!"

Question: {input}

{agent\_scratchpad}"""

prompt = ZeroShotAgent.create\_prompt(

tools,

prefix=prefix,

suffix=suffix,

input\_variables=["input", "agent\_scratchpad"],

)

memory = ConversationBufferMemory(memory\_key="chat\_history")

llm\_chain = LLMChain(llm=llm, prompt=prompt)

agent = ZeroShotAgent(llm\_chain=llm\_chain, tools=tools, verbose=True)

agent\_chain = AgentExecutor.from\_agent\_and\_tools(

agent=agent, tools=tools, verbose=True

)

agent\_chain.run("上海的人口是多少？")

> Entering new AgentExecutor chain...

Thought: I need to find the current population of Shanghai.

Action: Search

Action Input: "上海的人口是多少？"

Observation: 26.32 million (2019)

Thought:

I now know the current population of Shanghai.

Final Answer: The population of Shanghai is 26.32 million (2019).

> Finished chain.

从给出的答案“ 'The landmark building of "it" is the Empire State Building in New York City.' ”,我们可以对比得出，没有记忆组件的 Agent,面对“它”这样的指示代词无能无力，给出了莫名其妙的一个答案。它并不能关联上下文，推理出“它”指的是“上海”。

agent\_chain.run("它的地标建筑是什么？")

> Entering new AgentExecutor chain...

Thought: I need to find out the landmark building of "it".

Action: Search

Action Input: "it landmark building"

Observation: Landmark Builds, Joplin, Missouri. 186 likes · 1 talking about this. Engaging teens in local history and STEM education by tapping the creative power of ...

Thought:This search result is not relevant to the question. I need to refine my search query.

Action: Search

Action Input: "it city landmark building"

Observation: Landmark buildings are icons of a place. They create a statement about the city's legacy and influence how we think of that place. Landmark buildings stand out ...

Thought:This search result is still not relevant to the question. I need to refine my search query further.

Action: Search

Action Input: "it city famous landmark building"

Observation: Empire State Building, Manhattan, New York City, USA, the Art Deco skyscraper is New York's most popular landmark and a symbol for the American way of life, ...

Thought:

This search result is relevant to the question. The landmark building of "it" is the Empire State Building in New York City.

Thought: I now know the final answer.

Final Answer: The landmark building of "it" is the Empire State Building in New York City.

> Finished chain.

通过示例代码，展示了如何向 Agent 添加记忆组件，以及记忆组件在 Agent 中的作用。

## 6.4 内置记忆组件的对比

在进行长期对话时，由于语言模型可接受的令牌范围有限，我们可能无法将所有的对话信息都包含进去。为了解决这个问题，LangChain 提供了多种记忆组件，下面，我将通过一些代码示例来阐释我们在使用这些不同记忆组件时的区别。

首先，我们需要了解的是，LangChain 提供了包括聊天窗口缓冲记忆类、总结记忆类、知识图谱和实体记忆类等多种记忆组件。这些组件的不同主要体现在参数配置和实现效果上。选择哪一种记忆组件，需要根据我们在实际生产环境的需求来决定。

例如，如果你与聊天机器人的互动次数较少，那么可以选择使用 ConversationBufferMemory 组件。而 ConversationSummaryMemory 组件不会保存对话消息的格式，而是通过调用模型的摘要能力来得到摘要内容，因此它返回的都是摘要，而不是分角色的消息。

ConversationBufferWindowMemory 组件则是通过设置参数 k 来决定保留的交互次数。例如，如果我们设置 k = 2，那么只会保留最后两次的交互记录。

此外，ConversationSummaryBufferMemory 组件可以设置缓冲区的标记数 max\_token\_limit ，在做摘要的同时记录最近的对话。

ConversationSummaryBufferWindowMemory 组件既可以做摘要，也可以记录聊天信息。

实体记忆类是专门用于提取对话中出现的特定实体和其关系信息的。知识图谱记忆类则试图通过对话内容来提取信息，并以知识图谱的形式呈现这些信息。

值得注意的是，摘要类、实体记忆、知识图谱这些类别的记忆组件在实现上相对复杂一些。例如，实现总结记忆的时候，需要先调用大模型得到结果，然后再将这个结果作为总结记忆的内容。

在实体记忆和知识图谱这些类别的记忆组件中，返回的不是消息列表类型，而是可以格式化为三元组的信息。

我们在使用这些记忆组件的时候，最为困难的部分就是编写提示词模板。所以，如果你想要更深入地理解和学习这两种记忆组件，那么就需要特别关注他们的输出类型和提示词模板。

## 6.4.1 总结和缓冲区总结记忆

在进行长期的对话时，由于语言模型可接受的令牌范围有限，我们可能无法将所有的对话信息都包含进去。为了解决这个问题，LangChain 提供了不同的总结记忆组件来控制提示词的长度。接下来，我将通过一些代码示例来阐释我们在使用这些不同记忆类型时的区别。

首先，我们需要了解的是，LangChain 提供了两种主要的总结记忆组件：会话缓冲区总结记忆（ConversationSummaryBufferMemory）和会话总结记忆（ConversationSummaryMemory）。这两者的定义是什么呢？

会话总结记忆是一种总结记忆组件，它不会逐字逐句地存储我们的对话，而是将对话内容进行摘要，并将这些摘要存储起来。这种摘要通常是整个对话的摘要，因此每次我们需要生成新的摘要时，都需要对大语言模型进行多次调用，以获取响应。

而会话缓冲区总结记忆则结合了会话总结记忆的特性和缓冲区的概念。它会保存最近的交互记录，并将旧的交互记录编译成摘要，同时保留两者。但与会话总结记忆不同的是，会话缓冲区总结记忆是使用令牌长度而非交互次数来决定何时清除交互记录。这个记忆组件设定了缓冲区的标记数 max\_token\_limit，超过此限制的对话记录将被清除。

两种类型的公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。 这里我们使用的链组件只是一个简单的对话链 ConversationChain，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。：

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

             temperature=0,

             max\_tokens = 256)

from langchain.chains import ConversationChain

会话总结记忆

我们先看会话总结记忆, 这里先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationSummaryMemory

memory = ConversationSummaryMemory()

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new  chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

The human introduces themselves as "美丽", to which the AI responds with a friendly greeting and informs them that they are an AI who can both answer questions and converse. The AI then asks the human what they would like to ask, to which the human responds by asking the AI what their mood is. The AI responds that they are feeling good and are excited to be learning new things, and then asks the human what their mood is. The human then requests to find the customer service center, to which the AI responds that they can help them find it and asks where they want to find the customer service center.

Human: 我的洗衣机坏了

AI:

'> Finished chain.

你会看到，Current conversation 它不会逐字逐句地存储我们的对话，而是将对话内容进行摘要，并将这些摘要存储起来。这种摘要通常是整个对话的摘要。

会话缓冲区总结记忆

运行完公共代码后，再导入且实例化组件。我们设置为缓冲区的标记数 max\_token\_limit 限制为 40 个标记。

From langchain.chains.conversation.memory import ConversationSummaryBufferMemory

memory = ConversationSummaryBufferMemory(llm=OpenAI(),max\_token\_limit=40)

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    memory=memory

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new  chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

System:

The AI is introduced and greets the human, telling the human that it is an AI and can answer questions or have a conversation. The AI then asks the human what they would like to ask, to which the human replied asking how the AI is feeling. The AI replied that it was feeling good and was excited for the conversation. The human then asked to find the customer service center, to which the AI replied that it could help the human find the customer service center, and asked if the human wanted to know the location.

Human: 我的洗衣机坏了

AI:

'>  Finished chain.

'很抱歉听到你的洗衣机坏了。我可以帮助你找到客户服务中心，你可以联系他们来解决你的问题。你想知道客户服务中心的位置吗？'

你会看到，Current conversation 内容丢失了前面对话内容，但做了摘要，将对话以摘要的形式其包含在内，比如丢失了打招呼的对话。在最后给出了范围在 40 个数量以内的标记的对话记录, 即保留了 “Human: 我的洗衣机坏了” 。

但是在这里好像并没有对我们控制提示的长度有明显的改善，只不过是把所有的对话编程了摘要，从结果上来看，似乎比对话更啰嗦，使用的单词数量更多。

更新摘要

我们接着与他对话，不同的是我们抛出更多的问题，之前说洗衣机坏了，现在我抛出一些干扰的对话，比如说手机也坏了。我们看看他的摘要是否会更新？

# 请依次执行而不是一次性执行

conversation.predict(input="你知道洗衣机的操作屏幕显示ERROR是怎么回事吗?")

conversation.predict(input="我不知道他们的位置，你可以帮我找到他们的位置吗？")

conversation.predict(input="我打过他们客服中心的电话了，但是没人接听？")

conversation.predict(input="我的手机也坏了")

最后我们看看记忆内存中保存了什么：

print(memory.moving\_summary\_buffer)

The AI is introduced and greets the human, telling the human that it is an AI and can answer questions or have a conversation. The AI then asks the human what they would like to ask, to which the human replied asking how the AI is feeling. The AI replied that it was feeling good and was excited for the conversation. The human then asked to find the customer service center, to which the AI replied that it could help the human find the customer service center and asked if the human wanted to know the location. The human then stated that their washing machine and phone were broken, to which the AI apologized and offered to help the human find the customer service center so they can contact them to solve their problem, asking if the human wanted to know the address of the customer service center. The AI then offered to provide the address of the customer service center for the human, asking if they wanted to know it and offering to help the human find other contact methods such as email or social media accounts.

我们可以看到记忆存的是“The human then stated that their washing machine and phone were broken”, 多次对话的摘要，更新了。很早之前我们说洗衣机坏了，但是对话结束说手机坏了。这个组件做总结记忆的时候，将两个相同的事情合并了。

总结记忆组件的优势

我们可以看到，会话总结记忆和会话缓冲区总结记忆在实现方式上有着显著的差异。由于会话总结记忆是基于整个对话生成的，所以每次进行新的摘要调用时，我们需要对大语言模型进行多次调用，以获取对话的摘要。

这两种总结记忆组件在对话管理中起着重要的作用，特别是在对话的 token 数超过模型能够处理的范围时，这种摘要能力就显得尤为重要。

无论是对话长度较长，还是需要进行精细管理的情况，会话总结记忆和会话缓冲区总结记忆都能够提供有效的帮助。

## 6.4.2 会话记忆和窗口记忆

为了比较 ConversationBufferMemory 和 ConversationBufferWindowMemory 之间的区别。我们先导入公共的代码。

公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。 这里我们使用的 Chain 只是一个简单的对话链 `ConversationChain`，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。：

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

temperature=0,

max\_tokens = 256)

from langchain.chains import ConversationChain

缓冲区记忆

我们先看对话缓冲区记忆, 这里先导入且实例化 ConversationBufferMemory 组件。

from langchain.chains.conversation.memory import ConversationBufferMemory

memory = ConversationBufferMemory()

我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: 你好，我叫美丽

AI: 你好，美丽！很高兴认识你！我是一个AI，我可以回答你的问题，也可以与你聊天。你想问我什么？

Human: 今天心情怎么样？

AI: 今天我的心情很好！我很开心能够和你聊天！

Human: 我想找客户服务中心

AI: 好的，我可以帮助你找到客户服务中心。你知道客户服务中心在哪里吗？

Human: 我的洗衣机坏了

AI: 哦，很抱歉听到你的洗衣机坏了。你可以联系客户服务中心来获得帮助。你知道客户服务中心的联系方式吗？

Human: 我的洗衣机坏了

AI:

'> Finished chain.

你会看到，Current conversation 内部把所有人类和 AI 的对话记录都保存了。这样做可以让你看到我们之前对话的确切内容，这是 LangChain 中最简单的记忆形式，但仍然非常强大，特别是当你知道人们与你的聊天的互动次数有限，或者你实际上要在五次互动后关闭它，诸如此类的情况下，这种记忆将非常有效。

窗口缓冲区记忆

运行完公共代码后，再导入且实例化窗口缓冲区记忆ConversationBufferWindowMemory 类。

from langchain.chains.conversation.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=2)

conversation = ConversationChain(

llm=llm,

verbose=True,

memory=memory

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了")

执行完最后一条对话后，我们看到的会话链显示：

'> Entering new chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: 今天心情怎么样？

AI: 今天我的心情很好！我很开心能够和你聊天！

Human: 我想找客户服务中心

AI: 好的，我可以帮助你找到客户服务中心。你知道客户服务中心在哪里吗？

Human: 我的洗衣机坏了

AI:

'> Finished chain.

缓冲区与窗口缓冲区记忆的区别

为了对比两种类型的区别，我们采用的对话的内容是一样，但是由于我们设置了 k = 2，但你实际上可以将其设置得比这个高得多，可以获取最后 5 次或 10 次互动，具体取决于你想使用多少个标记。

最后打印的 Current conversation 部分，我们可以看到最初打招呼，介绍我自己的信息和 AI 回应并没有记忆下来。它丢失了“你好，我叫美丽”这句话，它只将最后两次互动传递给大型语言模型。

所以如果你有一些情况，可以将其设置为 k = 5 或者 k = 10 ，大多数对话可能不会有很大变化。实际上缓冲区窗口记忆比缓冲区记忆多了限制，不会记录所有人和 AI 的会话记录, 而是根据配置 K 的数值来对话记录条目，实现控制提示长度的目的。

## 6.4.3 实体和知识图谱记忆组件

在处理复杂对话时，我们常常需要提取对话中的关键信息。这种需求促使我们发展出了知识图谱和实体记忆这两种主要工具。

知识图谱是一种特殊的记忆类型，它能够根据对话内容构建出一个信息网络。每当它识别到相关的信息，都会接收这些信息并逐步构建出一个小型的知识图谱。与此同时，这种类型的记忆也会产生一种特殊的数据类型——知识图谱数据类型。

另一方面，实体记忆则专注于在对话中提取特定实体的信息。它使用大语言模型(LLM)提取实体信息，并随着时间的推移，通过同样的方式积累对这个实体的知识。因此，实体记忆的结果通常表现为关于特定事物的关键信息字典。

这两种工具都试图根据对话内容来表示对话，并提取其中的信息。实体记忆和知识图谱记忆就是这种情况的两个例子。

为了让你能更清晰地理解它们的功能，我们在代码示例中放入了一些简单的提示。这些提示能够让你看到当 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉时，它们实际上在做什么。我们的目标是让 AI 更专注于我们所说的内容，而不是无端地添加新的信息。

在实践中，我们通常会使用简单的提示来引导 AI 的操作。当 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉时，这些提示才会被使用。通过这种方式，我们可以确保 AI 在处理对话时始终保持清晰和准确。

公共代码

先安装库：

!pip -q install openai langchain

设置密钥：

import os

os.environ['OPENAI\_API\_KEY'] = ''

引入各组件，实例化一个会话链（ConversationChain）。为了让你能更清晰地理解它们的功能，我们在引入了提示模板组件，构造一个提示，方便大家理解。

这里我们使用的 Chain 只是一个简单的对话链 `ConversationChain`，允许我们跟 OpenAI 模型交互，并传递我们想要说的内容。

from langchain import OpenAI

llm = OpenAI(model\_name='text-davinci-003',

             temperature=0,

             max\_tokens = 256)

from langchain.chains import ConversationChain

from langchain.prompts.prompt import PromptTemplate

知识图谱记忆代码

先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationKGMemory

构建一个简单的提示, 让 AI 仅使用相关信息部分中包含的信息，并且不会产生幻觉。通过这种方式，我们可以确保 AI 在处理对话时始终保持清晰和准确。

template = """The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context.

If the AI does not know the answer to a question, it truthfully says it does not know. The AI ONLY uses information contained in the "Relevant Information" section and does not hallucinate.

Relevant Information:

{history}

Conversation:

Human: {input}

AI:"""

prompt = PromptTemplate(

    input\_variables=["history", "input"], template=template)

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    prompt=prompt,

    memory=ConversationKGMemory(llm=llm)

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了,操作面板出现ERROR字样")

conversation.predict(input="我的保修卡编号是A512423")

执行完最后一条对话后，我们打印记忆组件保存的知识图谱数据的结果：

print(conversation.memory.kg)

print(conversation.memory.kg.get\_triples())

我们可以看到记忆内存里面保存了对话中的关键信息。

<langchain.graphs.networkx\_graph.NetworkxEntityGraph object at 0x000001C953D48CD0>

[('AI', 'good mood', 'has a'), ('AI', 'new skills', 'is learning'), ('AI', 'talking to Human', 'enjoys'), ('Customer Service Center', 'city center', 'is located in'), ('Customer Service Center', '24 hour service', 'provides'), ('Customer Service Center', 'website', 'can be found on'), ('Customer Service Center', 'phone', 'can be contacted by'), ('Washing machine', 'ERROR on the control panel', 'has'), ('Human', 'A512423', 'has a warranty card number')]

实体知识记忆

先导入且实例化组件。

from langchain.chains.conversation.memory import ConversationEntityMemory

from langchain.chains.conversation.prompt import ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE

我们引入 Langchain 封装好的实体记忆的提示模板 ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE。

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    prompt=ENTITY\_MEMORY\_CONVERSATION\_TEMPLATE,

    memory=ConversationEntityMemory(llm=llm)

)

让我们开始对话，每次输入后等待 AI 返回的信息后，再输下一条：

# 请依次运行以下代码，不要一次性运行。

conversation.predict(input="你好，我叫美丽")

conversation.predict(input="今天心情怎么样？")

conversation.predict(input="我想找客户服务中心")

conversation.predict(input="我的洗衣机坏了,操作面板出现ERROR字样")

conversation.predict(input="我的保修卡编号是A512423")

执行完最后一条对话后，我们打印记忆组件保存的知识图谱数据的结果：

print(conversation.memory.entity\_cache )

我们可以看到记忆内存里面保存了对话中的关键信息。

['A512423', 'ERROR']

这两种工具都试图根据对话内容来表示对话，并提取其中的信息。

## 6.5 本章小结

本章我们深入探讨了记忆组件在大语言模型中的重要性及其应用。首先，我们阐述了大语言模型无法记忆对话历史的问题，以及记忆组件如何帮助解决这一问题。随后，我们对记忆组件的基本概念和工作机制进行了深入的介绍，让读者能够理解记忆组件的核心功能和工作原理。

在讨论了记忆组件的理论基础之后，我们进一步探讨了记忆组件在实际应用中的价值，特别是在聊天机器人或聊天代理中的作用，以及如何通过记忆组件优化聊天体验。我们通过具体的源码示例展示了如何实例化和使用记忆组件，进一步帮助读者掌握记忆组件的实际操作。

我们还详细介绍了如何在具有“外部数据”检索器的数据连接模块LEDVR工作流中，将记忆组件与链组件组合起来，实现数据检索增强效果，从而提升QA问答应用的回答质量。同时，我们还通过对比分析了各种会话场景下的内置记忆组件的实现代码，让读者更好地理解各个组件之间的区别和联系。

另外我们还展示了如何让智能体 Agent 拥有记忆，使用了工具组件，记忆组件，链组件等完成更复杂的任务。

总的来说，本章的目标是帮助读者全面理解记忆组件的重要性、工作原理和实际应用，以及如何通过自定义记忆组件来满足特定的需求。我们希望读者通过学习本章的内容，能够更好地理解和掌握记忆组件。